but it can also run on servers using environments like **Node.js**. JavaScript supports **object-oriented**, **functional**, and **event-driven** programming paradigms, making it a powerful tool for both frontend and backend development.

---

2. 🔁 What is the difference between `var`, `let`, and `const`?

Choosing the right keyword for declaring variables prevents bugs related to scoping, hoisting, and immutability. Interviewers ask this to assess how well you understand modern ES6+ JavaScript.

Answer:

- `var` ➡ **Function scoped**, hoisted and can be re-declared within the same scope. It was commonly used before ES6.
- `let` ➡ **Block scoped**, not re-declarable within the same scope, and it's hoisted but not initialized (TDZ - temporal dead zone).
- `const` ➡ **Block scoped**, cannot be reassigned (immutable binding), though objects and arrays declared with `const` can still be mutated.

Use `let` and `const` for cleaner, more predictable code. `const` is generally preferred unless the value needs to change.

---

3. 🏷️ What are data types in JavaScript?

Knowing how data is represented and behaves in JavaScript helps prevent type-related bugs and improves debugging skills.

Answer:
JavaScript has two main categories of data types:

- **Primitive types** (immutable, stored by value):
  `string`, `number`, `boolean`, `null`, `undefined`, `symbol`, `bigint`

- **Non-primitive types** (mutable, stored by reference):
  `object`, `array`, `function`

Understanding the difference between primitive and non-primitive types is crucial when working with assignments, comparisons, and memory management.

---

4. 🧮 What is the difference between `==` and `===`?

This question tests your grasp of JavaScript's type coercion system, which can often lead to subtle bugs if misunderstood.

Answer:

- `==` ➡ Performs **type coercion** before comparison (loose equality). Example: `'5' == 5` → `true`
- `===` ➡ Compares both **value and type** (strict equality). Example: `'5' === 5` → `false`

Always use `===` to avoid unexpected behavior due to type coercion. It ensures cleaner and more predictable comparisons.

---

5. 📦 What is hoisting in JavaScript?

Hoisting is a fundamental concept that affects how variables and functions are accessed in different scopes. Interviewers use this to test your understanding of execution context.

Answer:
Hoisting is JavaScript's default behavior of **moving variable and function declarations to the top of their scope** before code execution.

- Variables declared with `var` are hoisted and initialized with `undefined`.
- `let` and `const` are hoisted too, but they remain uninitialized in the **Temporal Dead Zone (TDZ)**.
- Function declarations are fully hoisted, meaning you can call them before their actual definition in the code.

Understanding hoisting helps prevent reference errors and improves code clarity.

Final Round AI
Fast Track Your Career
With Final Round AI, your path to a job offer is faster than ever.
From personalized coaching to AI-powered tools, we make job
searching smarter and more efficient.

Try Final Round AI for FREE today! 🔥

6. 🔐 What are closures in JavaScript?

Closures are a core concept in JavaScript and are heavily used in callbacks, currying, data privacy, and functional programming patterns.

Answer:
A closure is created when a function "remembers" the variables from its outer lexical scope even after that outer function has finished executing.
Example:

```
function outer() {
  let counter = 0;
  return function inner() {
    counter++;
    return counter;
  };
}
const count = outer();
count(); // 1
count(); // 2
```

⛶
Here, `inner` still has access to `counter`, even after `outer` has run.

---

7. 🕐 What's the difference between synchronous and asynchronous code?

JavaScript is single-threaded. Understanding async behavior is crucial for building performant web apps.

Answer:

- **Synchronous code** blocks further execution until it completes, it runs line by line.
- **Asynchronous code** runs in the background (e.g., HTTP requests, timers), allowing the program to continue executing other tasks. It uses callbacks, Promises, or `async/await` to handle execution flow.

---

8. ➡ What are arrow functions?

Arrow functions are concise and behave differently in terms of `this`. Great for functional-style programming.

Answer:
Arrow functions provide a shorthand syntax (`() => {}`) and don't bind their own `this`, `arguments`, `super`, or `new.target`. This makes them ideal for short functions and

callbacks but not suitable as methods or constructors.

---

9. 🗂️ What is lexical scope?

Understanding lexical scope helps you reason about variable visibility and how closures work.

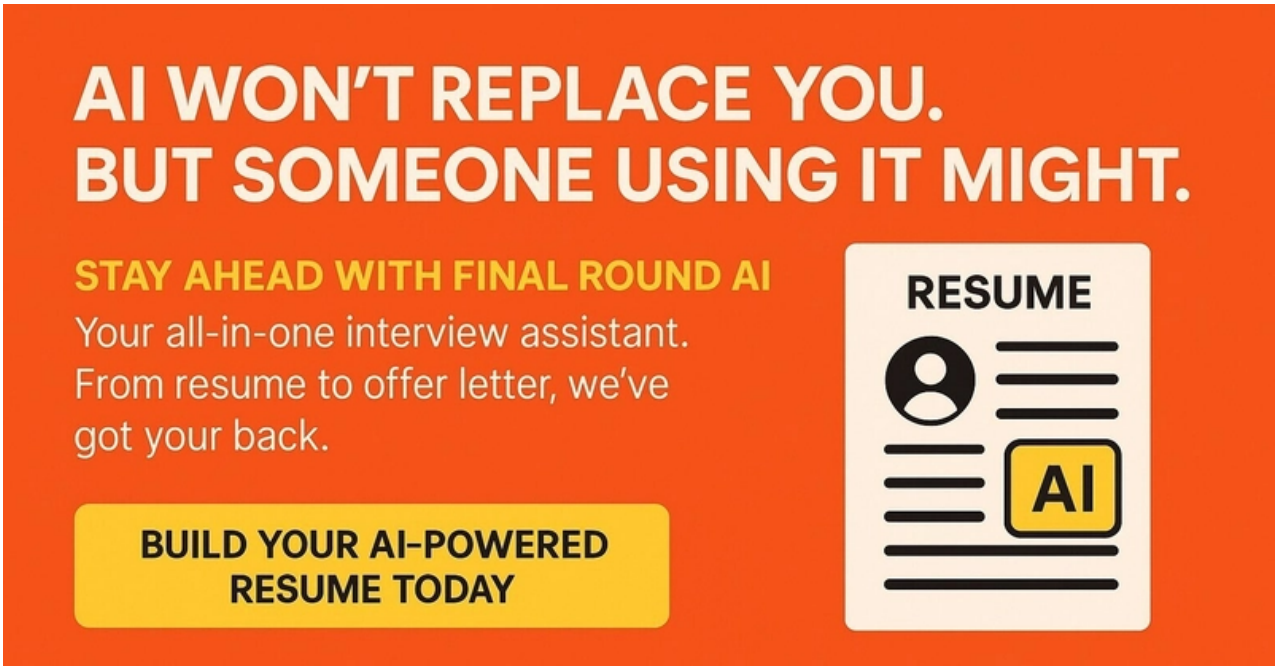
Answer:
Lexical scope means that the scope of a variable is determined by its position in the source code. Inner functions have access to variables declared in outer scopes. It's also the reason closures can access variables even after their parent function has returned.

---

10. 🌀 What is the event loop?

It's the heart of JS's async behavior. Knowing this will help you avoid race conditions and improve performance.

Answer:
The **event loop** manages the execution of multiple chunks of your program over time. It moves tasks between the **call stack** and the **callback/task queue**. It ensures non-blocking behavior by running tasks when the call stack is empty.



11. 🏴 How do you clone an object?

Copying objects is common, but you need to know when it's shallow vs deep to avoid reference bugs.

Answer:

- **Shallow copy**: `Object.assign({}, obj)` or `{...obj}`

- **Deep copy**: `JSON.parse(JSON.stringify(obj))` (note: loses functions, dates, and special types)
- Advanced: Use libraries like Lodash's `cloneDeep()` for deep cloning complex objects.

---

12. 🧾 What's the difference between `map()`, `filter()`, and `reduce()`?

These are fundamental tools in working with arrays and data transformations.

Answer:

- `map()`: Transforms each item and returns a new array.
- `filter()`: Filters items based on a condition.
- `reduce()`: Accumulates values into a single result (e.g., sum, object merging).

---

13. 🕵️ How do you check if a value is an array?

Arrays and objects are both of type `"object"`, so you need a reliable method.

Answer:
Use `Array.isArray(value)` for an accurate check. Avoid using `typeof`, which returns `"object"` for arrays.

---

14. 💣 What is destructuring in JavaScript?

Destructuring makes your code cleaner and easier to read.

Answer:
Destructuring lets you unpack values from arrays or properties from objects into distinct variables.

```
const [a, b] = [1, 2];
const { name } = { name: "Alice" };
```

This simplifies data access from nested structures and APIs.

---

15. 🔗 What is the spread operator?

Spread is used in everything, cloning, merging, function calls, etc.

Answer:
The spread operator (`...`) allows you to expand elements of an iterable (like arrays or objects):

```
const arr = [1, 2];
const newArr = [...arr, 3]; // [1, 2, 3]
```

It's useful for immutability patterns in React and merging data.

Try Final Round AI for FREE today! 🔥

16. 🧵 What are promises in JavaScript?

Promises are essential for managing asynchronous operations in a cleaner, more readable way than callbacks.

Answer:
A Promise represents the result of an async operation and has three states: `pending`, `fulfilled`, and `rejected`.
Instead of deeply nested callbacks (callback hell), promises use `.then()` and `.catch()` for chaining:

```
fetch('api/data')
  .then(res => res.json())
  .then(data => console.log(data))
  .catch(err => console.error(err));
```

17. 💤 What is async/await?

`async/await` simplifies working with promises, making async code look and behave more like synchronous code.

Answer:
`async` functions return a promise, and `await` pauses execution until that promise resolves or rejects.
This makes code cleaner and easier to follow:

```
async function getData() {
  try {
    const res = await fetch('api/data');
    const data = await res.json();
    console.log(data);
  } catch (error) {
    console.error(error);
  }
}
```

18. 📡 What's the use of `fetch()`?

`fetch()` is a modern standard for making network requests.

Answer:
`fetch()` is a browser API for making HTTP requests and returns a Promise. It's used for GET, POST, PUT, DELETE requests, and can be combined with `async/await`.

---

19. 🚨 How do you handle errors with async/await?

Async code is more error-prone. Graceful error handling is critical.

Answer:
Use `try...catch` blocks around `await` calls:

```
try {
  const res = await fetch(url);
  const data = await res.json();
} catch (err) {
  console.error("Failed to fetch:", err);
}
```

This avoids unhandled promise rejections.

---

20. 🔪 What is `Promise.all()`?

It's a key technique for parallel async operations like multiple API calls.

Answer:
`Promise.all()` takes an array of Promises and resolves when all succeed, or rejects if any fail. Useful for parallel loading:

```
await Promise.all([fetchA(), fetchB(), fetchC()]);
```

NEVER FREEZE UP IN AN INTERVIEW AGAIN
Final Round AI's Interview Copilot gives you real-time suggestions during interviews.
TRY INTERVIEW COPILOT FOR FREE TODAY!

21. 🧱 What are JavaScript modules?

Modules let you organize and reuse code across files.

Answer:
JS Modules use `export` to share code and `import` to consume it. They support encapsulation and are widely used in modern development (ES6+, React, Node.js, etc.).

---

22. 🕳️ What's the difference between `null` and `undefined`?

These are frequent sources of bugs and confusion.

Answer:

- `undefined`: A variable that has been declared but not assigned.
- `null`: An explicit assignment indicating "no value".

Use `null` intentionally; avoid leaving values `undefined`.

---

23. 📈 Explain debounce and throttle.

These techniques optimize performance, especially in event-heavy UIs.

Answer:

- **Debounce**: Waits until a function hasn't been called for X ms before running it (e.g., search input).
- **Throttle**: Ensures a function runs at most once per X ms (e.g., scroll events).

24. 🪚 What is the `this` keyword?

Misunderstanding `this` causes many bugs in object-oriented JS.

Answer:
`this` refers to the object that owns the current code. In methods, it refers to the parent object. In regular functions, `this` depends on the call context (or is `undefined` in strict mode). Arrow functions do **not** have their own `this`.

25. 🌳 What's prototypal inheritance?

It's how JavaScript implements OOP, not with classes (until ES6), but prototypes.

Answer:
In prototypal inheritance, objects inherit properties from other objects via their prototype chain. It's a flexible and dynamic alternative to classical inheritance.

Final Round AI
Real-Time AI Support for Your Tech Interview Success
More than just preparation – Interview Copilot actively supports you during your live interview, ensuring you never feel stuck.
Get hired at top tech companies!

26. 🧩 What is the DOM?

Frontend developers constantly manipulate the DOM to update the UI.

Answer:
The **Document Object Model (DOM)** is a tree-like structure representing a web page's HTML elements. JavaScript can access and manipulate this structure dynamically.

27. 🕵️ What's the difference between `==` and `===` in DOM comparison?

Comparing DOM elements can yield unexpected results if types mismatch.

Answer:
Same rule applies: use `===` for strict equality. When comparing DOM nodes, `node1 === node2` checks if both refer to the **exact same element**.

---

28. 📚 How do you select elements in the DOM?

Selecting elements is the first step in any DOM manipulation.

Answer:

- `document.getElementById('id')`
- `document.querySelector('.class')`
- `document.querySelectorAll('div')`

These methods help you target and modify elements dynamically.

---

29. 🖱️ What is event delegation?

It improves performance by attaching fewer event listeners.

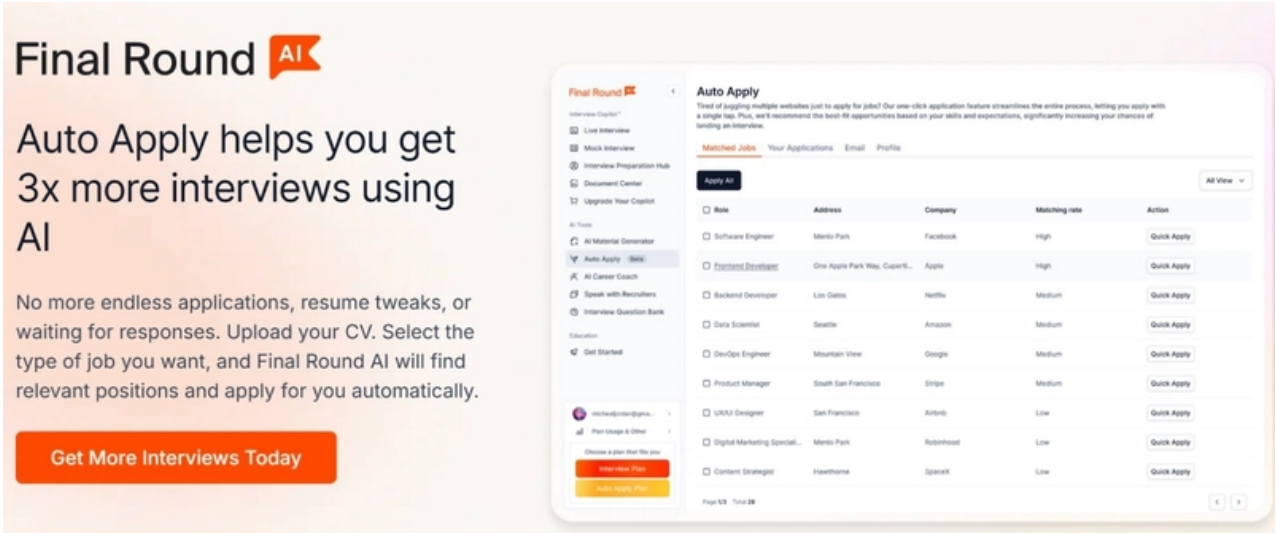
Answer:
Event delegation uses **bubbling** to catch events higher up the DOM tree. For example, attach one click listener to a parent, rather than many to each child.

---

30. 🚗 How do you prevent default behavior in an event?

Preventing default behavior is often necessary for form handling or custom interactions.

Answer:
Call `event.preventDefault()` inside your event handler to stop default browser behavior (like form submission or link navigation).


Final Round AI
Auto Apply helps you get 3x more interviews using AI
No more endless applications, resume tweaks, or waiting for responses. Upload your CV. Select the type of job you want, and Final Round AI will find relevant positions and apply for you automatically.
Get More Interviews Today
Auto Apply

31. 📜 What are template literals?

They make string formatting cleaner and more dynamic.

Answer:
Template literals use backticks and support interpolation:

```
const name = "Bob";
console.log(`Hello, ${name}!`);
```

They also support multi-line strings.

---

32. 📞 What is a callback function?

Callbacks are foundational to async JS, event handling, and array methods.

Answer:
A **callback function** is passed as an argument to another function and executed later. It enables things like event handlers and async logic.

---

33. ❌ What are the falsy values in JavaScript?

Conditional logic often depends on truthy/falsy checks.

Answer:
Falsy values are: `false`, `0`, `''` (empty string), `null`, `undefined`, `NaN`. All others are truthy. These influence conditions and short-circuit logic.

---

34. ⚖️ Difference between `typeof` and `instanceof`?

Type checking helps with debugging and enforcing logic.

Answer:

- `typeof`: Returns a string describing the type (`'object'`, `'function'`, `'string'`, etc.).
- `instanceof`: Checks if an object inherits from a constructor's prototype.

---

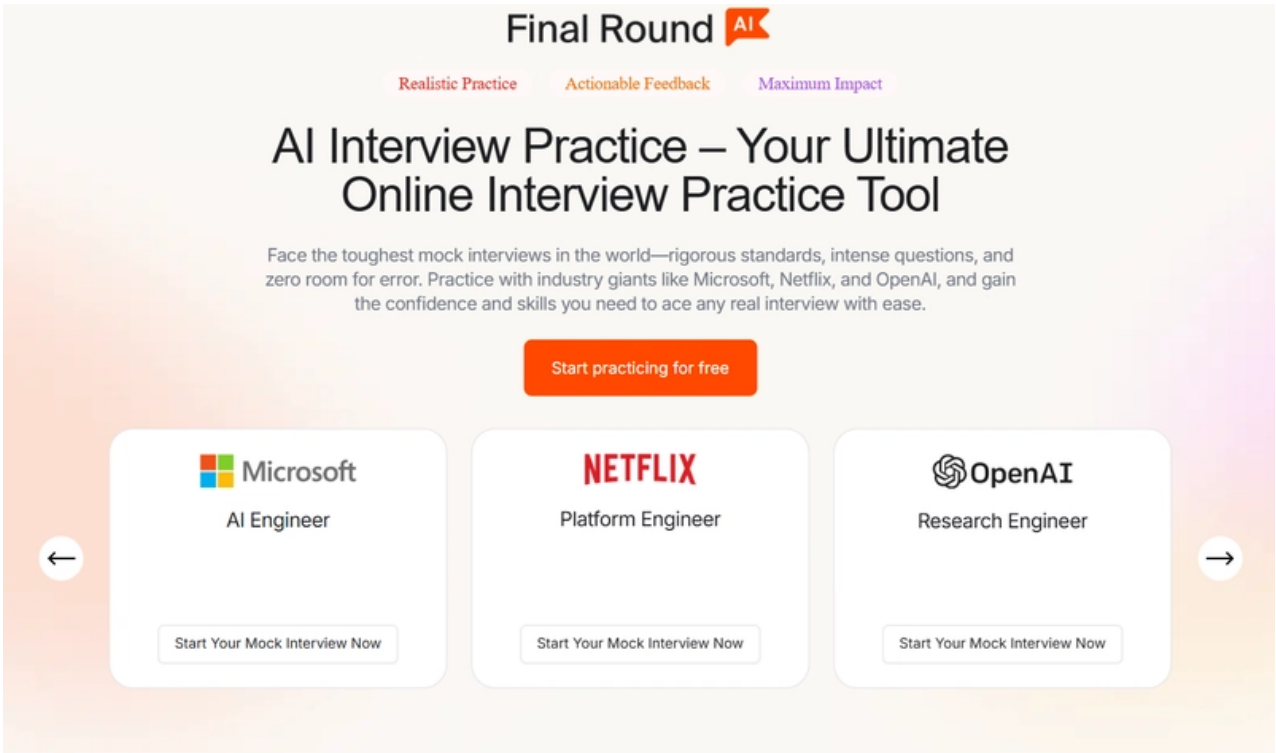
35. 🔥 What are immediately invoked function expressions (IIFE)?

Useful for avoiding polluting global scope and creating isolated code blocks.

Answer:
An **IIFE** runs immediately after it's defined:

```
(function() {
  console.log("Runs immediately!");
})();
```

Great for modules and closures.


Final Round AI
Realistic Practice
Actionable Feedback
Maximum Impact
AI Interview Practice – Your Ultimate Online Interview Practice Tool
Face the toughest mock interviews in the world—rigorous standards, intense questions, and zero room for error. Practice with industry giants like Microsoft, Netflix, and OpenAI, and gain the confidence and skills you need to ace any real interview with ease.
Start practicing for free
Microsoft
AI Engineer
Start Your Mock Interview Now
NETFLIX
Platform Engineer
Start Your Mock Interview Now
OpenAI
Research Engineer
Start Your Mock Interview Now

36. 🗃️ What's the difference between shallow and deep copy?

Copying data incorrectly leads to reference bugs.

Answer:

- **Shallow copy**: Copies top-level properties, references nested objects.
- **Deep copy**: Recursively copies everything. Required when you want full data isolation.

---

37. 🗑️ How does garbage collection work in JavaScript?

Knowing memory management helps avoid leaks and performance issues.

Answer:
JS uses automatic garbage collection. When no references to an object remain, it becomes unreachable and is removed from memory by the GC.

---

38. 🌐 What is the difference between `localStorage`, `sessionStorage`, and `cookies`?

Knowing where and how to store client data securely and appropriately is a critical frontend skill.

Answer:

- `localStorage`: persistent storage, up to 5MB, doesn't expire.
- `sessionStorage`: temporary, cleared when the tab is closed.
- `cookies`: small (4KB), sent with every request, can be accessed by server.

```
localStorage.setItem("user", "Alice");
```

Use cookies for authentication, `localStorage` for preferences, and `sessionStorage` for temporary data.

---

39. 💭 What is a service worker?

It's the backbone of Progressive Web Apps (PWAs).

Answer:
A **service worker** is a JS script that runs in the background and enables offline access, push notifications, and background sync. It intercepts network requests and can cache assets for offline use.

---

40. 🧗‍♂️ What are higher-order functions?

JavaScript is a functional language at heart. This concept underpins much of modern JS (like `map`, `filter`, etc.).

Answer:
A higher-order function is a function that takes another function as an argument or returns one.